COARA: Code Offloading on Android with RMI and AspectJ

Nir Shmuel Hauser
COARA: Code Offloading on Android with RMI and AspectJ

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Nir Shmuel Hauser

Submitted to the Senate
of the Technion — Israel Institute of Technology
Adar 5774 Haifa February 2014
This research was carried out under the supervision of Prof. Roy Friedman, in the Faculty of Computer Science.

Acknowledgements

I would like to thank Professor Roy Friedman for his guidance throughout this challenging process. I would not have been able to complete the work without him keeping me on track and providing insightful ideas.

I would like to thank Daniel Hurwitz, Jimy Pesin, Neer Roggel, Rouven Strauss, and Eric Yudin for providing input on my thesis. I would like to especially thank Eitan Rosenfeld for the days he invested to make my final draft the best it could be, and for the dozens of hafouchs that we drank while working together. It helped immensely.

Finally I would like to thank Guy, Hagar, and my parents for their assistance as I worked through this process. I literally could not have done it without your unwavering support.

The Technion’s funding of this research is hereby acknowledged.
# Contents

List of Figures

<table>
<thead>
<tr>
<th>Abstract</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2 Related Work</td>
<td>5</td>
</tr>
<tr>
<td>3 Background</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Android</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Aspect-Oriented Programming (AOP) and AspectJ</td>
<td>7</td>
</tr>
<tr>
<td>3.3 Java Remote Method Invocation (RMI)</td>
<td>8</td>
</tr>
<tr>
<td>3.4 Java Annotations</td>
<td>8</td>
</tr>
<tr>
<td>4 Design Goals and Architecture</td>
<td>9</td>
</tr>
<tr>
<td>4.1 Code transfer and registration</td>
<td>10</td>
</tr>
<tr>
<td>4.2 Method identification</td>
<td>10</td>
</tr>
<tr>
<td>4.3 Method offload and state transfer</td>
<td>11</td>
</tr>
<tr>
<td>4.4 Serialization Wrappers</td>
<td>12</td>
</tr>
<tr>
<td>4.5 Object Proxies</td>
<td>12</td>
</tr>
<tr>
<td>4.6 Decision Engine</td>
<td>17</td>
</tr>
<tr>
<td>4.7 Object Cache</td>
<td>17</td>
</tr>
<tr>
<td>4.8 Alternative Code execution</td>
<td>18</td>
</tr>
<tr>
<td>4.9 COARA Build process</td>
<td>19</td>
</tr>
<tr>
<td>4.10 Running Android on the server</td>
<td>20</td>
</tr>
<tr>
<td>4.11 Error handling</td>
<td>20</td>
</tr>
<tr>
<td>4.12 Security</td>
<td>21</td>
</tr>
<tr>
<td>5 Evaluation</td>
<td>23</td>
</tr>
<tr>
<td>5.1 Methodology</td>
<td>23</td>
</tr>
<tr>
<td>5.2 Case Studies</td>
<td>23</td>
</tr>
<tr>
<td>5.2.1 Pocket Chess</td>
<td>24</td>
</tr>
<tr>
<td>5.2.2 Linpack Benchmark</td>
<td>25</td>
</tr>
<tr>
<td>5.2.3 Face Detection</td>
<td>27</td>
</tr>
<tr>
<td>5.2.4 The Amount of State Transfer</td>
<td>31</td>
</tr>
</tbody>
</table>
List of Figures

4.1 COARA architecture ....................................................... 10
4.2 OffloadingAspect.aj code snippet ........................................... 11
4.3 OffloadingServer.java code snippet ......................................... 12
4.4 Eager Transmission .......................................................... 14
4.5 Lazy Transmission ........................................................... 15
4.6 Pipelined Transmission ....................................................... 16
4.7 Alternative Code Execution code snippet ................................. 18
4.8 Ant file code snippet ........................................................ 19
4.9 COARA architecture with Android running on server ..................... 20

5.1 Chess Performance .......................................................... 24
5.2 Chess Energy Consumption .................................................. 24
5.3 Linpack Benchmark Performance ............................................ 25
5.4 Linpack Benchmark Energy Consumption ................................... 25
5.5 Face Detection with JJIL ....................................................... 27
5.6 Face Detection Execution Times ............................................. 28
5.7 Face Detection Energy Consumption ....................................... 28
5.8 1/10 Face Detection Performance .......................................... 30
5.9 1/10 Face Detection Energy Consumption ................................. 30
Abstract

Smartphones suffer from limited computational capabilities and battery life. A method to mitigate these problems is code offloading: executing application code on a remote server. We introduce COARA, a middleware platform for code offloading on Android that uses aspect-oriented programming (AOP) with AspectJ. AOP allows COARA to intercept code for offloading without a customized compiler or modification of the operating system. COARA requires minimal changes to application source code. Since state transfer to the server is often a bottleneck that hinders performance, COARA uses AOP to intercept the transmission of large objects from the client and replaces them with object proxies. The server can begin execution of the offloaded application code, regardless of whether all required objects been transferred to the server. We run COARA with Android applications from the Google Play store on a Nexus 4 running unmodified Android 4.3 to prove that our platform improves performance and reduces energy consumption. Our approach yields speedups in real world applications of 24x and 6x over WiFi and 3G respectively.
Chapter 1

Introduction

Mobile computing is quickly becoming the prominent computing and communication platform. Smartphones and tablets have become ubiquitous, while mobile applications are increasing in complexity. However, even with recent improvements in mobile processing power, server processors are substantially more powerful [Fli12]. The primary focus of mobile design is long battery life, minimum size and weight, and heat dissipation. Computing power is often compromised in favor of these other priorities [HPR+12]. New mobile devices such as the Pebble watch and Google Glass may have even less computing power than current smartphones [FMH13].

This challenge has been tackled in the past using the client-server model, where the developer is responsible for writing code that issues requests to an API on a server. The downside of this approach is that rather than focusing on application functionality, the developer must invest resources in writing code to support network communication. If network connectivity is poor or unavailable, the application may be unable to execute.

One approach to addressing these challenges is code offloading: automatically executing application code on a remote server. By identifying computation intensive code to be offloaded, we can leverage the disparity between mobile and server computing power to achieve better performance and longer battery life on the mobile device.

In this paper, we introduce COARA, a middleware platform for code offloading on Android that uses aspect-oriented programming (AOP) [KLM+97] with AspectJ [asp]. AOP allows COARA to intercept code for offloading without a customized compiler or modification of the operating system. COARA requires minimal changes to application source code. An object cache eliminates the need to resend unmodified objects in subsequent method calls.

Execution of a method on a remote server requires that the application state must be transferred along with the method. State transfer to the server is often a bottleneck that hinders performance, especially on low-bandwidth 3G connections. To minimize state transfer, COARA uses AOP to intercept the transmission of large objects from the client to the server, replacing the objects with object proxies. Object proxies act as server-side placeholders for objects whose state, by design, has not been transferred to the server. The server can begin execution of the offloaded application code, regardless of whether all required objects have been transferred to the server.

COARA enables two data transmission strategies that use object proxies: lazy and pipelined transmission. With lazy transmission, COARA assumes that large objects will not be needed and therefore replaces the objects with proxies when transferring state to the server. If code executing
on the server accesses the proxy, COARA will halt execution and grab the object from the mobile device. With pipelined transmission, COARA also replaces objects with proxies however operates under the assumption that most of the objects will be accessed. Therefore, immediately after offloading the method, COARA transmits the large objects to the server in a pipelined manner. Pipelined transmission allows the server to get a “head start” and execute code while the rest of the heap is still being transferred. COARA also supports an eager transmission strategy that transfers the entire reachable heap to the server without the use of object proxies. We discuss these strategies in greater detail in Section 4.5 and show their effectiveness in improving performance in our evaluation.

We evaluated COARA using a Nexus 4 running unmodified Android 4.3 on applications from the Google Play store as well as applications we wrote that wrap open source Java libraries. Our approach achieves speedups of 24x and 6x over WiFi and 3G respectively. While this paper focuses on performance improvement, COARA was able to achieve reductions of energy consumption of 11x over WiFi and 3x over 3G.

The rest of this paper is organized as follows. We discuss related work in Section 2 and present some background terminology and technology in Section 3. The design goals and architecture of COARA are explained in Section 4. We evaluate the usage and performance of COARA in Section 5. We discuss limitation of the current system and future work in Section 6 and conclude in Section 7.
Chapter 2

Related Work

FarGo [HBSG99] is a system that enables distributed applications to maintain a separation between application logic and the layout of components on the underlying physical system. FarGo allows components known as complets to be relocated based on events at runtime. FarGo extends Java and uses Java RMI to communicate between JVMs. While FarGo is able to decouple application logic from layout logic, the system is tightly coupled with the application itself. FarGo requires that existing applications be rewritten to adhere to the FarGo programming model.

Early attempts at offloading like Spectra [FNS01] and Chroma [BSPO03] focused on partitioning the application into modules and calculating the optimal offloading strategy. These partitioning schemes require significant modification of the original application.

The MAUI [CBC+10] architecture provides the capability of fine-grained code offload with minimal application changes. It shows that minimizing the size of application state transfer vastly improves performance by allowing more methods to be offloaded. MAUI minimizes state transfer by only sending the difference in application state on subsequent method call invocations. However, MAUI serializes the application state using XML which is much slower than binary serialization and results in data that is much larger [HJR+03]. MAUI uses a custom compiler to insert offloading logic into an application, which generates two separate code bases — one for the mobile device and one for the server. Custom compilation may complicate debugging for the application developer.

CloneCloud’s [CIM+11] approach is to migrate the Virtual Machine (VM) on which the application is running. It migrates the VM from an Android phone to a server. Unlike other approaches, CloneCloud does not require intervention on the part of the developer for offloading since it happens at the operating system level. It also has the ability to offload native method calls. The downside of CloneCloud’s approach to offloading is the overhead required to migrate the entire VM from the mobile device to the server. In addition, the server must already be running a hardware simulator with the same configuration as the mobile device, further complicating matters.

ThinkAir [KAH+12], like MAUI, provides method level code offloading but focuses more on scalability issues and parallel execution of offloaded tasks. It uses a custom compiler to insert offloading logic. ThinkAir creates a VM on the cloud and can allocate multiple VMs to handle a single application. However, the authors do not discuss how they merge the application states
from multiple VMs back to the original application.

The Cuckoo [KPKB12] framework takes advantage of the existing activity/service model in Android. The model makes a separation between the computation intensive parts of the application and the rest of the application. Cuckoo offloads the well-defined computation intensive portions of the applications. It requires the developer to write offloadable methods twice — once for local computations and once for remote computations. This allows flexibility but may lead to unnecessary code duplication.

COCA [CLC12] uses AspectJ to offload Android applications to the cloud. COCA finds that the overhead incurred by AspectJ is minimal. However, COCA is only capable of offloading pure functions which only access the function parameters and are therefore oblivious to the rest of the heap. Therefore COCA does not tackle the hard problem of state transfer which MAUI demonstrates is one of the key challenges of offloading. Requiring real world applications to only offload pure functions would severely limit the number of functions that could be offloaded. On the other hand, COARA supports state transfer and provides techniques that leverage AspectJ to minimize the resulting overhead.

Calling the Cloud [GRJ+09] is a middleware platform that can automatically distribute different layers of an application between the phone and the server, and optimize a variety of objective functions. However, it requires the application to be partitioned into several interconnected software modules using the R-OSGi module management tool. The authors expect that application developers could take up to a month to adapt their applications to be compatible with their solution.

COMET [GJM+12] is a runtime system that performs offloading with Distributed Shared Memory (DSM). DSM provides a virtual shared memory space that is accessible by threads on different machines without any work on the part of the developer. The advantage of this approach is that fine-grained parallel algorithms can be offloaded to multiple machines, resulting in improved performance. The downside of the DSM approach is that it can lead to large unnecessary state transfers. For example, COMET requires 4.5MB of state transfer to execute a chess game. For this reason, COMET is largely unsuccessful in improving performance over low-bandwidth 3G networks. Recent attempts at distributed memory, such as IBM’s X10 programming language [CGS+05], have forced the developer to be conscious of any remote calls in order to avoid this problem. COARA does not currently support offloading parallel algorithms.

Much of the work described in this section focuses on algorithms that decide when to offload and when to execute on the mobile device. In this paper we focus on how to offload code, and less on when to offload. However, COARA does provide a simple decision engine that decides when methods should be offloaded. Naturally, it can be extended with more sophisticated mechanisms.
Chapter 3

Background

3.1 Android

Android is a Linux-based operating system designed primarily for touchscreen mobile devices. Android applications are mostly developed in the Java language using the Android Software Development Kit (SDK) and compiled to Java bytecode. They are then converted from Java Virtual Machine-compatible class files to Dalvik-compatible DEX (Dalvik Executable) files that run on the Android Dalvik Virtual Machine (DVM). During the Android application build process those DEX files are bundled into an APK (Android Application Package) file. While not all Java bytecode can be translated to DEX, many Java libraries are compatible with Android.

3.2 Aspect-Oriented Programming (AOP) and AspectJ

AOP is a programming paradigm that promotes modularity by allowing the separation of cross-cutting concerns [KLM+97]. Concerns are defined as wide-reaching functional or nonfunctional requirements, such as logging, security, or offloading which can be found across modules. Rather than scattering the code to address these concerns throughout the application, we handle them in special classes called aspects. Aspects can alter behavior in the base code by defining a dynamic set of locations in the base code, using a pointcut, along with the functionality, or advice, that should be added. Our primary cross-cutting concern is offloading the execution of certain methods.

AspectJ [asp] is a widely used aspect-oriented extension for the Java programming language. It is open source and is integrated into Eclipse. AspectJ is a natural fit for our architecture. COARA uses AspectJ to implement an OffloadingAspect that handles offloading. COARA identifies which method invocations it wants to intercept with the appropriate pointcut. The use of AspectJ allows COARA’s intervention in the application to be expressed by an extension of the Java language in a clear and elegant way.

An additional feature of AspectJ is inter-type declaration which enables aspects to add fields to classes and interfaces. COARA uses inter-type declaration to implement the object proxy transmission optimization by adding a unique global identifier field to objects represented by proxies.

While most Android applications don’t currently use AspectJ, enabling AspectJ in an existing
application is quick and easy. The AspectJ debugger supports tracing, enabling the application
developer to attain a detailed understanding of COARA’s intervention in the flow of execution. However, the application developer does not have to be familiar with AspectJ to integrate COARA into an application.

3.3 Java Remote Method Invocation (RMI)

Java RMI is a Java API that performs the object-oriented equivalent of remote procedure calls (RPC). RMI provides the ability to make calls from one JVM to another. Upon a remote method invocation, all relevant Java objects are automatically serialized and sent to the remote JVM. RMI allows COARA to send the state of the application from the Android device to the server using native Java binary serialization. Since Dalvik does not support Java RMI, COARA uses an open source lightweight implementation of RMI called lipermi [lip].

3.4 Java Annotations

A Java Annotation is metadata that can be added to Java source code. COARA allows the developer to specify which methods should be offloaded with the use of annotations. The developer simply tags a method with the annotation @RemotableMethod and a decision engine decides whether or not to offload the method at runtime. Similarly, a developer can tag a class with the @EnableProxy annotation to enable object proxies.
Chapter 4

Design Goals and Architecture

The primary goal of COARA is to improve performance by offloading intensive computations while maximizing usability and ease of use for the application developer. To this end we enumerate the following objectives:

• **Simplicity and extensibility of the framework** — COARA avoids custom compilation and low level operating system changes. Our offloading strategy is clearly expressed through AOP.

• **Minimal changes to existing code** — We do not require developers to make significant changes to existing applications, nor do we force new applications to adhere to strict requirements. Incorporating COARA into existing projects should be quick and easy.

• **Minimize state transfer** — The overhead of state transfer is a significant impediment to the success of offloading, especially over low-bandwidth 3G networks. COARA offers techniques to minimize state transfer and improve performance.

• **Unmodified Android source code** — The Android operating system is open source and therefore it is possible to modify the base code. We do not modify the base code to enable application developers to use COARA immediately on any Android device running a recent version of Android.

• **Alternative code execution on server** — In most cases the server executes the same code as the mobile device. However, in certain cases we want to increase the **fidelity** of the application by running a more powerful algorithm on a server than we would on the mobile device. COARA provides the application developer the option to easily specify alternative code.

• **Code privacy** — Some developers are not comfortable with sharing all of their class files with a remote server. COARA allows the developer to easily prevent unnecessary class files from being transferred to the server.
4.1 Code transfer and registration

The COARA architecture is illustrated in Figure 4.1. The COARA server runs within a Java Virtual Machine (JVM). When a COARA-enabled Android application starts up, the COARA client connects to a fresh instance of a JVM running the COARA Server. Since Java is a portable language, it does not matter what operating system or architecture the JVM is running on, or whether that operating system is running within a virtual environment on top of a hypervisor.

COARA ensures that the server executes the same application code as the mobile device. During the Android application build process, a JAR file containing the Java bytecode of the application is bundled within the APK (Android Application Package) file. When the application begins execution on the mobile device, the COARA client sends an MD5 hash of the contents of the JAR file to the server. If the COARA server sees the JAR has been previously registered, it loads the JAR. Otherwise, the server requests the JAR file from the client. When the server receives the JAR from the client, it loads the JAR and caches the JAR locally for future sessions. While this approach adds some overhead at startup, it ensures that the code on the client and server are identical. To reduce this overhead, it would be possible to introduce a code repository similar to the one found in [CBC+10].

4.2 Method identification

In order to identify candidate methods for offloading, COARA takes a similar approach to [CBC+10] and [KAH+12], and requires the application developer to annotate the offloadable methods with the annotation @RemotableMethod. When such a method is invoked on the client, execution is intercepted by the AOP framework and transferred to OffloadingAspect (see Figure 4.2) to perform the offload.

As an example, we have enabled COARA in the open source Google Play application Pocket...
public aspect OffloadingAspect {

    // Define the annotated methods to intercept (pointcut)
    pointcut executeRemotable(Object obj) :
        execution (@RemotableMethod *=.*(..)) && this(obj);

    // Define execution after interception (advice)
    Object around(Object obj) : executeRemotable(obj) {
        ...
        // Perform offload
        Response response = offloader.executeMethod(
            appState, obj, method, params);
        ...
    }

    ...
}

Figure 4.2: OffloadingAspect.aj code snippet

Chess For Android [poc]. The only changes we made to the source code were:

1. Modified Position and SimpleEngine to implement java.io.Serializable
2. Added the @RemotableMethod annotation to SimpleEngine.go()

@RemotableMethod
public String go() { ... }

4.3 Method offload and state transfer

In order to execute a method remotely, it is not necessary to transfer the entire heap. The application state which COARA must transfer to the server includes:

- static objects in the heap
- the method parameters
- the object on which the method is invoked, and any objects in the heap that are reachable by that object. We will refer to this collection of objects as the reachable heap.

Rather than always transferring all static objects, COARA allows the developer to list the objects that are needed specifically in a configuration file.

When a method annotated with @RemotableMethod is invoked, the invocation is intercepted by AspectJ. The COARA client then issues an RMI call to Offloader.executeMethod(..) on the COARA server. When the COARA server receives an offloading request via RMI, OffloadingServer (Figure 4.3) uses Java reflection to dynamically invoke the method. When method execution concludes, the COARA server sends the modified state back to the client. The COARA client then uses Java reflection to update the transferred objects.

We considered other serialization schemes in addition to native Java Serialization. We looked at the Kryo [kry], XStream [xst], and Gson [gso] serialization libraries. However, we found that
public class OffloadingServer implements Offloader {
    public Response executeMethod(AppState state, Object obj, MethodWrapper method, List<Object> params)
        throws IOException {
        // Invoke the offloaded method using Java Reflection
        Object returnObject = method.getMethod().invoke(obj, params.toArray());
        ...
    }
}

Figure 4.3: OffloadingServer.java code snippet

those libraries are better at handling pre-defined types. Since COARA needs to handle an object of any class, we found they did not meet our needs.

Native Java Serialization requires that every transferred object belong to a class that implements java.io.Serializable. While implementing this interface requires more work on the part of the application developer, it ensures safety. This requirement forces the application developer to be conscious of what objects are being transferred. Native Java Serialization also provides developers with the option of customizing the serialization of a class by implementing the readObject() and writeObject() methods on the class [Blo08]. We believe approaches that offload without any input from the developer can lead the developer to ignore the costs of transferring large objects, and leaves the developer without the ability to optimize such a transfer.

4.4 Serialization Wrappers

Some classes, such as those that are part of the Android SDK, cannot be modified to implement Serializable. In this case COARA offers the option of serialization wrappers. The application simply needs to define a class that implements the SerializationWrapper interface and register it in the COARA configuration file. Serialization wrappers enable externalization of the serialization logic.

4.5 Object Proxies

For large state transfers, the major components of the state may be large objects that represent media such as images or sounds. The time required to serialize and transfer the large objects may decrease the speedup gains from offloading, especially over 3G. COARA mitigates this problem with object proxies.

The proxy pattern is a software design pattern. A proxy is an object that acts as a surrogate or placeholder for another object. Proxies are used to defer the cost of an object’s creation until the object is needed [GHJV95]. In the case of offloading, the “cost” involves transferring objects from the mobile device to the server. The use of proxies enables the server to begin execution of the offloaded method, regardless of whether all the objects in the reachable heap have been transferred.

The application developer enables object proxies on a particular class by annotating the class
with \texttt{@EnableProxy}. Using AspectJ’s inter-type field declaration, COARA adds three fields to the annotated class:

- \texttt{guid} (Global Unique Identifier) — used by the server to request an object from the client
- \texttt{emptyContainer} (boolean) — true if the real object is currently not available in the server’s local memory
- \texttt{isInCache} (boolean) — true if the real object is currently available in the server’s object cache (described in Section 4.7).

When COARA decides to offload a method to the server, it must first serialize the reachable heap on the client (as described in Section 4.6). If during serialization an object whose class is annotated with \texttt{@EnableProxy} is present in the reachable heap, COARA may replace the object with a proxy — an empty object instantiated via the default constructor. The serialized heap containing the proxy is transferred to the server. When the server deserializes the heap, it will contain a proxy rather than the actual object. If code executing on the server accesses an object proxy, AspectJ will halt execution until COARA makes the object available. Once the real object is available on the server, COARA uses reflection to copy the real object’s fields into the proxy and normal execution resumes.

We consider three separate strategies when dealing with state transfer from the client to the server: \textit{eager transmission}, \textit{lazy transmission} and \textit{pipelined transmission}. Each strategy is described along with a figure that illustrates a different use case.
Figure 4.4: **eager transmission** —
1) The object $x$ contains objects $a$ and $b$.
2) $x$.$\text{foo}()$ is invoked on the client. $x$ is sent to the server.
3) $x$.$a$ is modified on the server.
4) $x$.$\text{foo}()$ returns and the state in the client is updated.

- **eager transmission** — The entire reachable heap is transferred to the server with the offloaded method. Proxies are not used. See Figure 4.4.
Figure 4.5: lazy transmission —
1) The object x contains objects x.a and x.b.
2) x.bar() is invoked on the client. Proxies are sent for x.a and x.b.
3) COARA halts execution on server and requests x.a from client.
4) The client sends x.a to the server.
5) Execution resumes on the server. x.a is modified.
6) x.bar() returns and the state in the client is updated.

- lazy transmission — Objects whose class is annotated with @EnableProxy are replaced with proxies during serialization on the client. The reachable heap (containing proxies) is transferred to the server with the offloaded method. If code executing on the server accesses a proxy, AspectJ halts execution and the COARA server issues a synchronous request for the object to the client. When the request completes, the object is loaded and execution resumes. See Figure 4.5.
• **pipelined transmission**—Objects whose class is annotated with `@EnableProxy` are replaced with proxies during serialization on the client. The reachable heap (containing proxies) is transferred to the server with the offloaded method. As soon as the method is offloaded, the client immediately initiates asynchronous transfers of the real objects to the server, one at a time. This allows code executing on the server to get a “head start” on computation without requiring all of the reachable heap to be present on the server. We call this strategy “pipelined” because the server is simultaneously computing while downloading object data for the next computation. In addition the client is simultaneously uploading object data while serializing objects for future upload. See Figure 4.6.

Lazy transmission is useful when there are many large objects in the reachable heap, but some may not accessed by code executing on the server. Pipelined transmission is useful when most of the large objects will be accessed by code executing on the server and the application developer has a general idea of the order in which they will be accessed.

COARA uses lazy and pipelined transmission only when transferring state from the client to the server. If COARA were to use these strategies when transferring state from the server to the client, a server failure would leave the client unable to continue execution because the client would be dependent on the server for a portion of the state.

In Section 5 we show that offloading with object proxies yields speedups of 24x for WiFi and...
6x for 3G and can decrease state transfer by over 90%.

4.6 Decision Engine

COARA offers a simple decision engine that decides whether to offload a specific method, and if so, which transmission strategy to use. The decision engine interfaces with a network profiler that periodically measures the bandwidth and latency of the network to allow better decision making. COARA’s architecture is designed to allow a developer to extend COARA and implement an alternative decision engine. In the future we plan to develop an advanced decision engine.

4.7 Object Cache

In order to minimize state transfer, COARA provides an object-level cache. When COARA transfers an object whose class is annotated with \texttt{@EnableProxy}, COARA caches a serialized version of the object at the client as well as the server. The next time COARA serializes the object for transfer, COARA compares the object with the previously stored copy. If the object has not been modified, COARA replaces the object with a proxy and sets the \texttt{isInCache} field to \texttt{true}. When the object is deserialized on the server, COARA will notice that the \texttt{isInCache} field is \texttt{true} and will retrieve the object from the server’s object cache and copy it into the proxy using reflection.

In our evaluation we find that the object cache achieves speedups of 31.9x over WiFi and 23.3x over 3G and can decrease state transfer by 99% in the best case scenario of 100% cache hit.
class MyClass{
    @RemotableMethod (
        alternativeClassName = "com.myApp.MyClass",
        alternativeMethodName = "executeAdvancedAlgorithm"
    )
    public void executeAlgorithm() // Runs on client
    { ... }

    public void executeAdvancedAlgorithm() // Runs on server
    { ... }
}

Figure 4.7: Alternative Code Execution code snippet

4.8 Alternative Code execution

Application fidelity can be defined as “the degree to which the results produced by an application match the highest quality results that would be produced given ample computational resources” [Fli12]. COARA enables application developers to designate alternative method implementations when executed on a remote server in order to increase fidelity.

For example, COARA can run a simple algorithm if the method is executing locally and a computationally expensive algorithm that provides higher quality results if it is executing on a remote server. The application developer enables this feature for a particular method by defining the optional elements alternativeClassName and alternativeMethodName in the @RemotableMethod annotation as demonstrated in Figure 4.7.
Table 4.1: config.xml parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_ip</td>
<td>string</td>
<td>offloading server IP address</td>
</tr>
<tr>
<td>server_port</td>
<td>int</td>
<td>offloading server port</td>
</tr>
<tr>
<td>static_classes</td>
<td>array</td>
<td>list of static classes to offload</td>
</tr>
<tr>
<td>class_proxies</td>
<td>array</td>
<td>serialization wrapper registration (see Section 4.4)</td>
</tr>
<tr>
<td>cache_enabled</td>
<td>boolean</td>
<td>object cache (see Section 4.7)</td>
</tr>
<tr>
<td>proxy_enabled</td>
<td>boolean</td>
<td>proxy server enabled</td>
</tr>
<tr>
<td>proxy_ip</td>
<td>string</td>
<td>proxy server IP address</td>
</tr>
<tr>
<td>proxy_port</td>
<td>int</td>
<td>proxy server port</td>
</tr>
</tbody>
</table>

Figure 4.8: build.xml Ant file

```xml
<project name="app" default="build-jar" basedir="." >
  <target name="build-jar" >
    <mkdir dir="res/raw" />
    <jar jar="res/raw/classes.jar" basedir="bin/classes"
         excludes="**/SecretClass.class"/>
  </target>
</project>
```

4.9 COARA Build process

We designed the COARA client to be easy to integrate into existing Android applications. The application developer has to follow a series of simple steps to enable COARA in Eclipse:

1. Install the AspectJ Eclipse plugin and enable AspectJ in the application.
2. Copy the AspectJ runtime JAR (aspectjrt.jar) to the libs folder.
3. Add the COARA JAR file to the project’s aspectPath.
4. Fill in a COARA XML configuration file (see Table 4.1).
5. Add the android.permissionINTERNET permission to the Android Manifest.
6. Add the provided COARA Ant file to the list of Builders for the project (Figure 4.8).

The Ant file allows exclusion of sensitive class files that the developer does not want to be offloaded via the exclude field.

After these steps are followed, the application developer enables offloading by designating at least one method with the @RemotableMethod annotation.
4.10 Running Android on the server

In Section 4.1, we indicated that the COARA server runs within a JVM. However, the JVM is unable to execute Android-specific classes which prevents some methods from being candidates for offloading. Therefore we have also successfully offloaded code to a COARA server running within an instance of Android-x86. We have written a COARA Server Android application which wraps the COARA Runtime and runs within the Dalvik VM as demonstrated in Figure 4.9.

Unlike [CIOa10] we did not succeed in leveraging the full potential of the server hardware with Android-x86 and therefore did not see significant performance improvement. As our team’s ability to extract better performance from Android-x86 improves, we will reconsider running the COARA server exclusively on Android-x86.

4.11 Error handling

COARA employs a simple timeout mechanism to handle server failures. If COARA detects a timeout, it simply runs the method on the local device. COARA has a Network Profiler that will attempt to reconnect with the server after a failure. Since any partial computations that may have happened on the server are not visible to the mobile device, there is no risk of corrupting data. The only risk is if the server executed a transaction on some other network resource which could be duplicated if it were executed again by the mobile device. Where such a risk exists, the developer should not allow COARA to offload the method.
4.12 Security

At the moment, COARA does not provide any security guarantees, therefore the server should be trusted by the client. In addition all data is transmitted without any encryption. However, COARA enables the developer to restrict which classes are offloaded to the server. Further, when the Android device is rooted, the COARA client can communicate with the server through an SSH tunnel. Adding direct support for encrypted communication using standard means such as SSL is one of our top priorities.
Chapter 5

Evaluation

5.1 Methodology

We evaluated the COARA client using an LG Nexus 4 with a quad-core 1.5 GHz Krait CPU running unmodified Android 4.3 JellyBean. For the offloading server, we used a Dell PowerEdge R210 II with an Intel Xeon E3-1220 V2 3.10GHz single-core CPU running Ubuntu 12.04.3 LTS with JVM 1.6. We demonstrate that COARA can achieve significant performance gains and energy savings using a commodity server.

For each evaluation workload, COARA used either 802.11g WiFi in the Taub Computer Science building at the Technion Israel Institute of Technology, or the 3G network of the HotNet cellular provider in Israel. Mobile devices do not need to be rooted to support COARA. However, we rooted our Nexus 4 in order to use SSH Tunnel [SSH] to access a server behind a firewall when testing with 3G.

To evaluate energy consumption, we measured the voltage and current approximately every 10 milliseconds programmatically. On the Nexus 4, the Android OS exposes these values through the filesystem in files called \textit{current\_now} and \textit{voltage\_now} in the /sys/class/power_supply/battery folder.

5.2 Case Studies

We evaluate the performance and energy consumption of COARA on four Android applications. Each application was analyzed separately as a case study to more effectively convey the benefits and shortcomings of offloading.
5.2.1 Pocket Chess

We evaluated the performance of Pocket Chess, an open source Android application available on the Google Play store [poc]. In order to be able to run the same game multiple times for this evaluation, we modified the source code of the chess game to remove any randomness in the chess engine. We then simulated the human moves using Android’s uiautomator testing framework [uia]. This resulted in a repeatable 30 move chess game.

While COARA succeeded in improving computation time to determine the next move, the execution time for the User Interface (UI) tasks remained constant as expected. In Figure 5.8 we break down the computation time by UI and Computation. COARA achieved overall game speedups of 3.9x over WiFi and 3.2x over 3G. If we focus solely on the computation portion, COARA achieved speedups of 9.7x over WiFi and 6.0x over 3G.

When evaluating battery consumption we did not break out the UI consumption. Figure 5.2 demonstrates that COARA achieved energy consumption improvements of 6.4x over WiFi and 3.1x over 3G.

Pocket Chess represents an ideal candidate for COARA due to the combination of high computation and small state transfer. From a qualitative perspective, without COARA there is noticeable lag whenever the opponent is deciding the next move. With COARA enabled, it appears to the end user as though the decision is instantaneous.
5.2.2 Linpack Benchmark

We evaluated the performance of the Linpack benchmark, available as part of the 0xbench Google Play application [0xba]. The benchmark measures the system’s floating point computing power by solving a dense $n \times n$ system of linear equations [0xbb].

The benchmark runs through a computation-intensive loop 10 times. Without enabling COARA, the benchmark runs for 11.8ms on the Nexus 4, resulting in 58.7 MFLOPS. Because of network latency, enabling COARA results in slowdowns of 0.8x for WiFi and 0.3x for 3G. To minimize the effect of network overhead, we ran the benchmark with a large number of iterations. The results appear in Figure 5.3. For 1000 iterations, we see speedups of 15.8x for WiFi and 13.3x 3G. The benchmark’s results demonstrate what our intuition tells us — the usefulness of offloading is highly dependent on the size of the computation.

When evaluating the effect of COARA on battery consumption we observed a similar trend. Figure 5.4 shows that at a low number of iterations, we find similar or worse battery consumption
with offloading. As the number of iteration grows, COARA achieves energy consumption improvements of 34.5x over WiFi and 15.6x over 3G.
5.2.3 Face Detection

Thus far we have only looked at applications with negligible state transfer. To demonstrate COARA’s ability to optimize state transfer, we implemented a Face Detection Android application (Figure 5.5) that uses Jon’s Java Image Library (JJIL) [jjil], an open source, pure Java image processing library.

We ran the Face Detection application with 1 and 10 images with eager, pipelined, and lazy transmission. When the cache is used, we preload the cache and conduct the tests with a best case 100% hit rate.

Performance of Face Detection with 1 Image

The best performance for processing 1 image without the cache is achieved by eager transmission which results in speedups of 8.1x for WiFi and 3.3x for 3G (see Figure 5.6). Eager transmission does not use object proxies. Since only one large object is being transferred, there is no benefit to the overhead of object proxies. We see the impact of this overhead with slightly lower speedups for lazy and pipelined transmission. We immediately see the benefits of the object cache with speedups of 16.6x for WiFi and 9.6x for 3G.

Performance of Face Detection with 10 Images

As COARA sends more large objects, eager transmission yields greater performance. Improvement occurs because the one time overhead of the initial offload is offset by the relatively larger gains from offloading. Lazy transmission does not see a significant improvement over 1 image because it now requires 10 calls from the server to client to retrieve each image, resulting in additional overhead.

The most dramatic improvement is seen by pipelined transmission, which is able to take full advantage of the 10 images and provides a speedup of 24.7x over WiFi and 5.1x over 3G. Another significant observation is that pipelined transmission achieves a speedup of 1.42x WiFi relative to
Figure 5.6: Absolute execution times for Face Detection with no offloading, WiFi offloading, and 3G offloading. Each bar represents the average of 10 trials. Computation speedup figures are shown in the table.

<table>
<thead>
<tr>
<th>FD 1 Image Speedup</th>
<th>WiFi</th>
<th>3G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eager</td>
<td>8.1x</td>
<td>3.3x</td>
</tr>
<tr>
<td>Lazy</td>
<td>7.3x</td>
<td>3.1x</td>
</tr>
<tr>
<td>Pipelined</td>
<td>6.4x</td>
<td>3.5x</td>
</tr>
<tr>
<td>Cache</td>
<td>16.6x</td>
<td>9.6x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FD 10 Images Speedup</th>
<th>WiFi</th>
<th>3G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eager</td>
<td>17.5x</td>
<td>4.8x</td>
</tr>
<tr>
<td>Lazy</td>
<td>11.1x</td>
<td>3.7x</td>
</tr>
<tr>
<td>Pipelined</td>
<td>24.7x</td>
<td>5.1x</td>
</tr>
<tr>
<td>Cache</td>
<td>31.9x</td>
<td>23.3x</td>
</tr>
</tbody>
</table>

Figure 5.7: Absolute energy consumption for Face Detection with no offloading, WiFi offloading, and 3G offloading. Each bar represents the average of 10 trials. Energy consumption improvement figures are shown in the table.

<table>
<thead>
<tr>
<th>FD 1 Image Energy</th>
<th>WiFi</th>
<th>3G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eager</td>
<td>7.3x</td>
<td>2.2x</td>
</tr>
<tr>
<td>Lazy</td>
<td>7.2x</td>
<td>1.7x</td>
</tr>
<tr>
<td>Pipelined</td>
<td>8.7x</td>
<td>2.3x</td>
</tr>
<tr>
<td>Cache</td>
<td>20.5x</td>
<td>6.1x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FD 10 Images Energy</th>
<th>WiFi</th>
<th>3G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eager</td>
<td>11.8x</td>
<td>3.3x</td>
</tr>
<tr>
<td>Lazy</td>
<td>9.2x</td>
<td>2.0x</td>
</tr>
<tr>
<td>Pipelined</td>
<td>11.3x</td>
<td>3.5x</td>
</tr>
<tr>
<td>Cache</td>
<td>25.9x</td>
<td>17.4x</td>
</tr>
</tbody>
</table>
eager transmission. We feel this makes a powerful argument for the use of pipelined transmission. However, pipelined transmission performed similarly to eager transmission over 3G. Again, the object cache sees the best performance with speedups of 31.9x for WiFi and 23.3x for 3G.

**Face Detection and Energy Consumption**

The results in Figure 5.7 demonstrate that offloading with COARA significantly reduces energy consumption by up to 11.3x over WiFi and 3.5x over 3G without a cache, and 25.9x over WiFi and 17.4x over 3G with a cache. However, we see no significant advantage with pipelined transmission over eager transmission in regards to energy consumption.
Figure 5.8: Absolute execution times for Face Detection where the reachable heap contains 10 images yet only one is accessed by the offloaded method. Absolute execution times with no offloading, WiFi offloading, and 3G offloading. Each bar represents the average of 10 trials. Computation speedup figures are shown in the table.

Why use lazy transmission?

In the previous examples, the offloaded method accessed all of the large objects in the reachable heap. However, this may not be the case in practice. In order to demonstrate this use case, we ran an experiment with 10 images in the reachable heap. However, in this experiment, the offloaded method performs face detection only on the 10th image in the array. We will call this use case “1/10” face detection. The results are available in Figure 5.9.

With offloading over 3G, eager and pipelined transmission incur serious slowdowns of 0.5x and 0.7x, respectively, eliminating the benefit from offloading. The slowdowns occur because COARA transfers 10 large images, but code executing on the server only accesses one of them.

However, lazy transmission achieves a speedup of 3.1x over 3G. This is due to the fact that lazy transmission only retrieves the image from the client once it is accessed by code executing on the server. In [GJM+12] most of the applications did not receive benefits from offloading over 3G because of large state transfer.

In terms of energy consumption, eager and pipelined transmission both fail to show any improvement over WiFi and 3G. However, lazy transmission improves energy consumption by
<table>
<thead>
<tr>
<th>Application</th>
<th>Upload</th>
<th>Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
<td>6.4</td>
<td>4.7</td>
</tr>
<tr>
<td>Chess (30 move game)</td>
<td>83.5</td>
<td>75.6</td>
</tr>
<tr>
<td>FD 1 Image Eager</td>
<td>196.7</td>
<td>51.5</td>
</tr>
<tr>
<td>FD 1 Image Lazy</td>
<td>196.7</td>
<td>51.4</td>
</tr>
<tr>
<td>FD 1 Image Pipelined</td>
<td>196.3</td>
<td>50.7</td>
</tr>
<tr>
<td>FD 1 Image Cache</td>
<td>1.3</td>
<td>2.7</td>
</tr>
<tr>
<td>FD 5 Images Eager</td>
<td>748.1</td>
<td>170</td>
</tr>
<tr>
<td>FD 5 Images Lazy</td>
<td>748.3</td>
<td>173</td>
</tr>
<tr>
<td>FD 5 Images Pipelined</td>
<td>749.3</td>
<td>168.1</td>
</tr>
<tr>
<td>FD 5 Images Cache</td>
<td>1.6</td>
<td>7.1</td>
</tr>
<tr>
<td>FD 10 Images Eager</td>
<td>1486.9</td>
<td>327.8</td>
</tr>
<tr>
<td>FD 10 Images Lazy</td>
<td>1493.6</td>
<td>337.3</td>
</tr>
<tr>
<td>FD 10 Images Pipelined</td>
<td>1495.6</td>
<td>327.7</td>
</tr>
<tr>
<td>FD 10 Images Cache</td>
<td>1.8</td>
<td>7.8</td>
</tr>
<tr>
<td>FD 1/10 Eager</td>
<td>1487.5</td>
<td>323.3</td>
</tr>
<tr>
<td>FD 1/10 Lazy</td>
<td>117</td>
<td>28.6</td>
</tr>
<tr>
<td>FD 1/10 Pipelined</td>
<td>1494.6</td>
<td>62.1</td>
</tr>
</tbody>
</table>

*FD = Face Detection

6.4x over WiFi and 1.8x over 3G.

We could have avoided this specific use case by modifying the application code to remove the unnecessary 9 images from the reachable state. However, sometimes it may be unclear which objects will be accessed by the offloaded method. For example, conducting a search on a series of sorted objects.

5.2.4 The Amount of State Transfer

Table 5.1 displays the amount of data transferred when running COARA. It displays the amount of data uploaded from the device to the server and the amount of data downloaded from the server back to the mobile device. We do not consider the transmission of the code at startup because in most cases the code will already be cached on the offloading server.

Benchmark and Chess have very low state transfer. We did not enable object proxies with these two applications. By having the application developer decide where to offload, we can reduce the number of unnecessary reachable objects in the heap leading to a smaller state transfer.

The table clearly illustrates the benefits of lazy transmission with the “1/10” face detection use case. While both eager and pipelined transmission uploaded nearly 1.5MB to the server, lazy transmission only uploaded 117KB — a reduction of over 90%. Similarly, the object cache kept uploads to the server below 2KB, thus reducing state transfer by up to 99%.

5.2.5 Alternative Code Execution

As discussed in Section 4.8, COARA provides the ability to increase fidelity by specifying alternative methods to run on the server. To demonstrate this feature, we wrote a simple Android application that computes $\pi$ to a large number of digits with Machin’s formula using code found
in a Java RMI tutorial [PiR].

Our application runs a method \( \text{pi}() \) that calculates \( \pi \) to 10,000 digits. Using COARA’s alternative method feature, we specified an alternative method \( \text{piServer}() \) that calculates \( \pi \) to 20,000 digits. COARA yielded a \( \sim 5 \times \) speedup over WiFi and 3G and a higher quality result of twice as many digits.
COARA currently does not handle transferring of object locks. This means that COARA assumes that only one method is offloaded at a time and that no other threads are running on the client in the meantime. If this assumption is violated, object locks may not be respected which can result in unexpected behavior or deadlock. COMET [GJM+12] solves this problem with Distributed Shared Memory (DSM).

COARA is currently unable to offload native methods. [CBC+10] and [GJM+12] have been able to accomplish this. COARA could be extended to support this functionality.

COARA could be extended to use static analysis to detect which objects in the reachable heap are not accessed by a method and automatically annotate their classes with @EnableProxy. Static analysis could also be used to detect at compile time which objects are in the reachable heap and therefore must belong to classes that are Serializable.

Currently our decision engine is very simple. We would like to develop an engine that determines not only when methods are offloaded, but also which offloading strategy to use.

We would like to explore offloading coarse-grained parallel algorithms where each machine operates on a copy of shared memory but is oblivious of changes on other machines. An example would be performing facial recognition on multiple images in parallel.
Chapter 7

Conclusion

In this paper we have introduced COARA, a middleware platform for code offloading on Android that uses aspect-oriented programming (AOP) with AspectJ. AOP allows COARA to intercept code for offloading without a custom compiler or modifying the operating system. COARA requires minimal changes to application source code.

Using AOP, COARA intercepts the transmission of large objects from the client and replaces them with proxies, thereby increasing performance, reducing energy consumption, and minimizing state transfer.

We have shown that our pipelined transmission strategy was able to provide speedups of 24x over WiFi and 5x over 3G relative to execution on the mobile device. In addition, pipelined transmission was able to achieve a 1.42x speedup over WiFi relative to eager transmission.

In one case study, we observed that using eager transmission with a large state transfer resulted in a slowdown of 0.5x over 3G. The lazy transmission strategy minimized the state transfer by 90% resulting in a speedup of 3x over 3G. Lazy transmission can allow gains from offloading where they would be impossible with a full state transfer.

The object cache eliminates the need to resend unmodified objects. When the cache hit rate is 100%, COARA can decrease state transfer by 99%, resulting in speedups of 31.1x over WiFi and 22.7x over 3G.

COARA demonstrates that AOP offers an elegant solution for offloading. Using AOP, we implement techniques that mitigate the overhead of state transfer, which is one of the biggest impediments to the success of offloading.
Bibliography


Google Pocket Chess, Pocket Chess

The problem is to find a sequence of moves that leads to a checkmate or a draw. The moves are represented as pairs of move numbers and move directions. The goal is to find a sequence of moves that leads to a checkmate or a draw.

To solve this problem, we can use a search algorithm. One way to do this is to use a breadth-first search (BFS) algorithm. At each step, we generate all possible moves from the current position and add them to the queue. We then remove the first move from the queue and repeat the process.

Here is an example of a possible sequence of moves that leads to a checkmate:

```plaintext
1. e4
2. e5
3. f6
4. g4
5. g5
6. h4
7. h5
8. g6
9. hxg6
10. hxg6
```

In this sequence, the white king is in check and cannot move, so the black king mate. The sequence is found by searching through the possible moves and checking if the king is in check at each step.

To implement this algorithm, we can use a data structure to store the positions and moves. We can use a list to store the moves and a dictionary to store the positions. We can also use a set to keep track of the positions that have already been visited.

Here is an example implementation of the algorithm in Python:

```python
def search_moves(position, moves):
    if position in visited:
        return None
    visited.add(position)
    for move in moves:
        new_position = position
        new_position[move[0]] = move[1]
        result = search_moves(new_position, moves)
        if result:
            return result
    return None

# Example usage:
start = (0, 0)
possible_moves = [((0, 1), 'e'), ((1, 0), 'f')]
result = search_moves(start, possible_moves)
print(result)
```

This algorithm can be used to find a sequence of moves that leads to a checkmate or a draw. It can be adapted to find a sequence of moves that leads to a draw as well.

In conclusion, the Pocket Chess problem is a challenging problem that requires careful planning and strategy. By using a search algorithm, we can find a sequence of moves that leads to a checkmate or a draw. This problem can be used to test and improve the performance of search algorithms.
Takzir

The field of mobile computing (mobile computing) has seen rapid growth in recent years. Mobile devices have become ubiquitous, providing users with constant connectivity and access to the internet. This has led to the development of a variety of applications and services that can be accessed on the go.

In this context, the thesis presents a new approach for code offloading (code offloading) to mobile devices. The proposed method, known as COARA (COARA), allows applications to offload some of their computational tasks to the cloud. This is achieved by using AspectJ (Aspect Oriented Programming, AOP) to intercept and modify the application code at runtime.

The main contributions of the thesis are:

1. A novel approach for code offloading to mobile devices using AOP.
2. A middleware-based solution that facilitates the dynamic generation of proxies for remote methods.
3. An implementation of the proposed method that demonstrates its effectiveness in improving the performance of mobile applications.

The thesis also discusses the challenges and limitations of mobile computing and proposes solutions to overcome them. The results of the thesis highlight the potential of mobile computing as a powerful tool for enhancing user experience and improving productivity.

ה(bundle) בקבוצה התאימה של הפרפסור או שני פרסים, בפסקיין לומדים הפרחים.

תודה

ה_bundle מועדה למסוריה על פרויקט מחקר זה.
העבורה קור ב Dmitry研发技術づなوت� ב solución ב אונמא
אנדרואיד רוסקטים

תובר על מחקר

לשם משלוח חלקי של הדיווח המעשה לקרות
מצטער למסיעו במעדן התשובה

ניר שמוואל האודור

רונת לעטי התכנית --- מיכל טאטלניי ולישראל
אדר תשע"ד חפירה פברואר 2014
העבירה קוד בדמカフェית תוד שימור ב’אווה
אנדרואיד ואספקטים

ניר שמאלה הואר